9>1a 10, perform compression on mnist dataset using auto encoder.

Aim: to compress MNIST hand written digit image into a lower-dimensional representation using a autoencoder.

Algorithm

1, load and normalize the MNIST dataset
2, Built an autoencoder neural network
3, train the autoencoder to minimize the difference between original and reconstructed image
4, use the encoder to compress the image
5, use the decoder to reconstruct the image

Pseudo code:

load MNIST dataset
Normalize images to range [0,1]

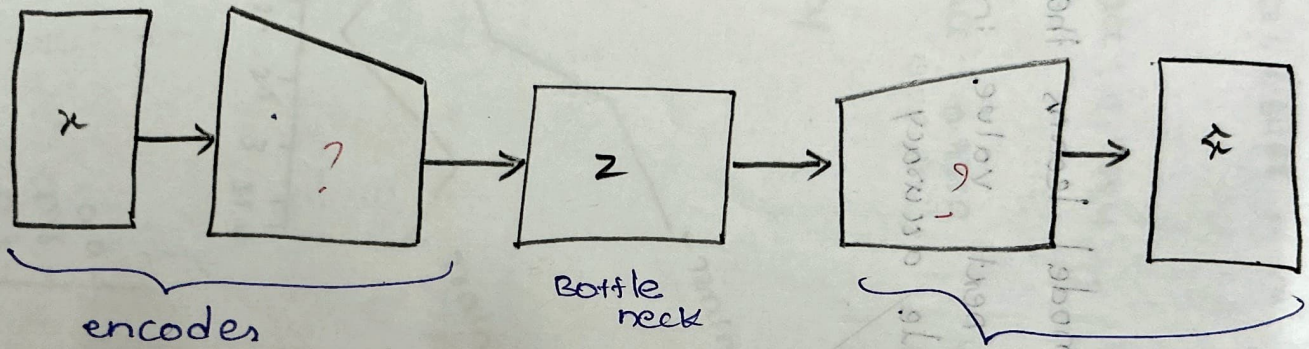
Define encoder network:

input : 28x28 image
Flatten input
Dense layers reducing dimension → latent space

Define decoder network:

input : latent vector
dense layer expanding dimension
Reshape output to 28x28 image

Autoencoder architecture



combine encoder with loss = mean-squared error.

train autoencoder on MNIST training images

For a given image:

compressed = encoder(image)
reconstructed = decoder(compressed)

Result:

typical reconstruction loss reduces significantly after ~~loss~~ training, showing the the auto encoder successfully

1, 12/6

~~combine encoder and decoder to form~~

output :

Epoch : [1|10] · loss :- 0.9496

Epoch : [2|10] loss : 0.9254

Epoch : [3|10] loss : 0.9254

Epoch : [4|10] loss : 0.9254

Epoch : [5|10] loss : 0.9254

Epoch : [6|10] loss : 0.9254

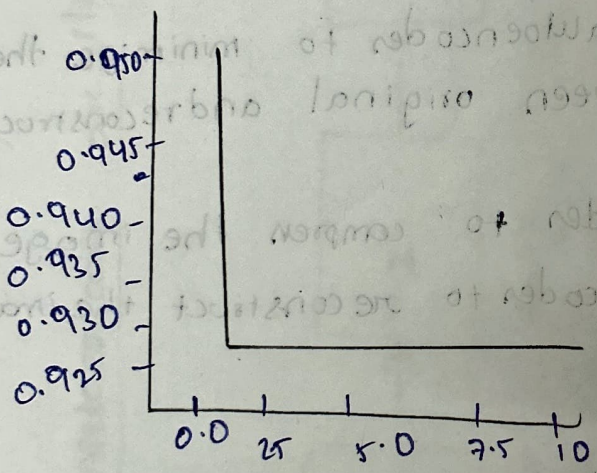
Epoch [7|10] loss :0.

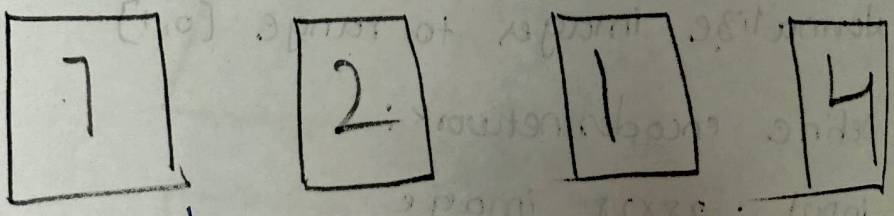
Epoch [8|10] loss: 0.

Epoch [9|10] loss :0.

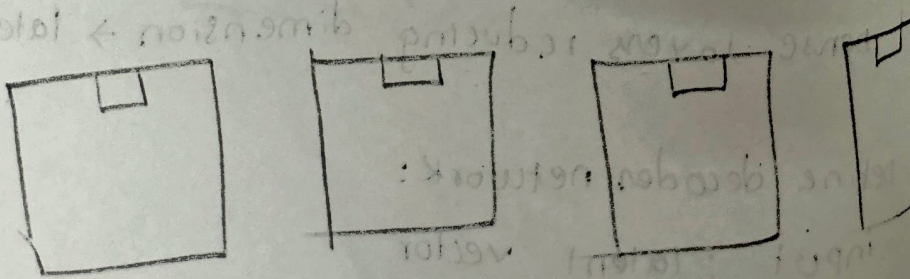
Epoch [10|10] loss:0.

Auto coder training loss



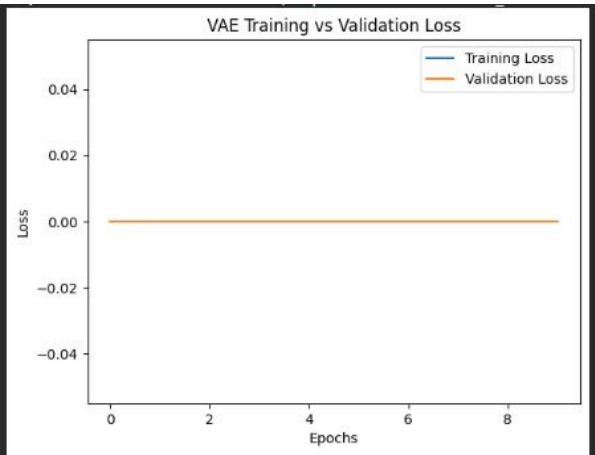


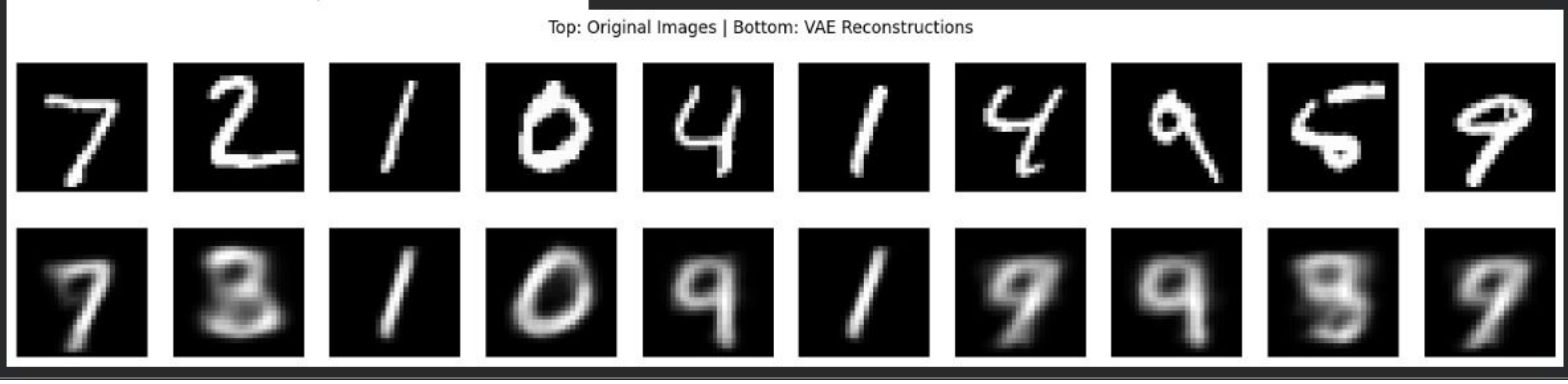
original images



Reconstructed images

```python
import tensorflow as tf
from tensorflow.keras import layers, Model
import matplotlib.pyplot as plt
import numpy as np

(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

latent_dim = 2
inputs = tf.keras.Input(shape=(784,), name="encoder_input")
x = layers.Dense(512, activation="relu")(inputs)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)

def sampling(args):
    z_mean, z_log_var = args
    epsilon = tf.random.normal(shape=tf.shape(z_mean))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon

z = layers.Lambda(sampling, name="z")([z_mean, z_log_var])
encoder = Model(inputs, [z_mean, z_log_var, z], name="encoder")
encoder.summary()

latent_inputs = tf.keras.Input(shape=(latent_dim,), name="z_sampling")
x = layers.Dense(512, activation="relu")(latent_inputs)
outputs = layers.Dense(784, activation="sigmoid")(x)
decoder = Model(latent_inputs, outputs, name="decoder")
decoder.summary()

class VAE(Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def call(self, inputs):
        z_mean, z_log_var, z = self.encoder(inputs)
        return self.decoder(z)

    def train_step(self, data):
        if isinstance(data, tuple):
            data = data[0]
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            # ✅ Fixed reconstruction loss
            reconstruction_loss = tf.reduce_mean(
                tf.keras.losses.binary_crossentropy(data, reconstruction) * 784
            )
            kl_loss = -0.5 * tf.reduce_mean(
                1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)
            )
            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.trainable_variables))
        return {"loss": total_loss}

    def test_step(self, data):
```


VAE Training vs Validation Loss
Training Loss
Validation Loss
0.04
0.02
0.00
−0.02
−0.04
Loss
0
2
4
6
8
Epochs


Top: Original Images | Bottom: VAE Reconstructions